Efficient QoS Routing
Stavroula Siachalou, Leonidas Georgiadis

Abstract—We consider the problem of routing in a network where QoS constraints are placed on network traffic. We provide two optimal algorithms that are based on determining the discontinuities of functions related to the optimization at hand. The proposed algorithms have pseudopolynomial worst case running time and for a wide variety of tested networks they have fairly satisfactory running times. They perform significantly better than the algorithm based on the direct application of the Dynamic Programming equations and can also be used in conjunction with known polynomial-time approximation algorithms to provide good average case behavior, in addition to guaranteeing polynomial worst-case running time.

Index Terms—Network Routing, QoS Routing, Graph theory, Simulations

I. INTRODUCTION
Transmission of multimedia traffic presents many challenges to the network designer. Voice and video packet streams require certain bandwidth as well as bounds on delay, loss probability and jitter in order to maintain reception quality. These issues give rise to the problem of routing multimedia traffic so that Quality of Service (QoS) is maintained. [11], [13],[14] otherwise known as the Constrained Shortest Path Routing Problem.

The QoS Routing Problem consists in finding an optimal-cost path from a source to a destination subject to one or more constraints (e.g., total delay and loss probability) on the path. It is well known that this problem is NP-complete [5] and several heuristics have been proposed for its solution [2], [3], [8]. On the other hand, fully polynomial ε-approximate solution schemes for the problem exist [7], [15]. These results were recently improved in [9] and were applied to related problems in [10]. In [6], polynomial algorithms have been proposed that find the optimal solution from one source to all destinations, within ε-deviation from the set path constraint. The algorithms in [6] and [9] use as a subroutine the iterations implied by the Dynamic Programming equation related to the problem at hand.

In this paper we provide two optimal algorithms for the QoS Routing problem. The algorithms consist in finding the discontinuity points of functions related to the optimization problem. Although pseudopolynomial, tests with a wide variety of networks, link costs and link constraints, show that the proposed algorithms have fairly satisfactory performance and can be used in practical systems. Moreover, if guaranteed polynomial worst-case running time is also desired, the algorithms can replace the dynamic programming recursions in the approximate polynomial-time algorithms in [6] and [9], to improve their average running time.

II. PROBLEM FORMULATION AND ALGORITHMS
Let G = (N, L) be a graph with node set N and link set L. A link with origin node m and destination node n is denoted by (m, n). With Np(n) and Nn(n) we denote the set of incoming and outgoing neighbors to node n, that is, respectively,

Np(n) = {m ∈ N : (m, n) ∈ L},
Nn(n) = {m ∈ N : (n, m) ∈ L}.

With each link l = (m, n), m, n ∈ N there is an associated cost cmn ≥ 0 and delay dmn ≥ 0. If p = (m1, ..., mk) is a directed path (a subgraph of G consisting of nodes m1, ..., mk, mi ≠ mj for all 1 ≤ i, j ≤ k, i ≠ j, and links (mi, mi+1), 1 ≤ i ≤ k − 1) then we define the cost and delay of the path respectively,

C(p) = Σ (m,n)∈p cmn,
D(p) = Σ (m,n)∈p dmn.

The set of all paths with origin node s, destination node n and delay less than or equal to d is denoted by Ps,n(d). The set of all paths from s to n is denoted simply by Ps,n. For any d, we are interested in finding a path p* ∈ Ps,n(d) such that

C(p*) ≤ C(p) for all p ∈ Ps,n(d).

Let Cn*(d) be the minimum of the costs of the paths p ∈ Ps,n(d). If Ps,n(d) = ∅, we define Cn*(d) = ∞. For node s, we also define

Cs*(d) = \begin{cases} ∞ & \text{if } d < 0 \\ 0 & \text{if } d ≥ 0 \end{cases}.

The algorithms to be presented below depend heavily on the properties of the functions Cn*(d). These properties are presented in the lemmas below.

Lemma 1: The functions Cn*(d), n ∈ N, n ≠ s, satisfy the following equations.

Cn*(d) = \min_{m∈N_p(n)} \{c_{mn} + C_m*(d - d_{mn})\}

Proof: These are the dynamic programming equations for the problem at hand [1, Problem 4.46.]. We only note that due to the fact that the links costs are nonnegative, we can use the equations as stated in the lemma instead of

Cn*(d) = \min_{m∈N_p(n)} \{C_n*(d - 1), \min_{m∈N_p(n)} \{c_{mn} + C_m*(d - d_{mn})\}\}.

S. Siachalou and L. Georgiadis are with Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki, Thessaloniki, Greece. E-mails: stavroula@psyche.ee.auth.gr, leonid@eng.auth.gr

IEEE INFOCOM 2003
0-7803-7753-2/03/$17.00 (C) 2003 IEEE
and we do not need to make any integrality assumptions on $d$.

**Lemma 2:** For any $n \in N$, $C_n^*(d)$ has the following properties

1) $C_n^*(d)$ is nonincreasing
2) If $C_n^*(d) < \infty$ for some $d$, then $C_n^*(d)$ is a piecewise constant, right continuous function of $d$. If $C_n^*(d)$ is discontinuous at $d_0$, then
   a) There is a path $p_d^*$ such that $C_n^*(d) = C(p_d^*)$ and $D(p_d^*) = d_0$.
   b) There is an $m \in N_+(n)$ such that $C_m^*(d)$ is discontinuous at $d_0 - d_{mn}$ and $C_m^*(d_0) = C_n^*(d_0 - d_{mn}) + c_{mn}$.
3) The set $D_n$ of points (real number pairs) $(d, C_n(d))$ at which discontinuities of $C_n(d)$ occur, is finite.

**Proof:**

1) The fact that $C_n^*(d)$ is nonincreasing follows directly from the definition.

2.a) Observe that if $C_n^*(d) < \infty$, then there must exist a path $p \in P_m(d)$ with $C(p) = C_n^*(d)$. Let $p_d^*$ be a path with smallest delay among the paths that have cost $C_n^*(d)$. By definition,

$$C_n^*(d) = C(p_d^*).$$

Moreover it holds that

$$C_n^*(d') > C_n^*(d)$$

for $d' < D(p_d^*)$.

To see this note that if $C_n^*(d') = C_n^*(d)$ then there must exist a path $q$ with delay at most $d' < D(p_d^*)$ such that $C(q) = C_n^*(d') = C_n^*(d)$, which contradicts the definition of $p_d^*$. Since $C_n^*(d)$ is nonincreasing, the only possibility is that $C_n^*(d') > C_n^*(d)$ for $d' < D(p_d^*)$.

Next, let $\tilde{p}_d$ be the path with smallest delay among the paths in the set

$$\mathcal{P}(n) = \{p \in P_m : C(p) < C_n^*(d)\}.$$

In case $\mathcal{P}(n) = \emptyset$ define $D(\tilde{p}_d) = \infty$. Then, it holds

$$C_n^*(d') = C(p_d^*)$$

for $D(p_d^*) \leq d' < D(\tilde{p}_d)$.

To see this note that if $C_n^*(d') = C(p_d^*)$ for some $d'$ such that $D(p_d^*) \leq d' < D(\tilde{p}_d)$, then there must exist a path $q$ with delay at most $d' < D(\tilde{p}_d)$ such that $C(q) = C_n^*(d') = C(p_d^*)$, which contradicts the definition of $\tilde{p}_d$. Since $C_n^*(d)$ is nondecreasing, the only possibility is that $C_n^*(d') = C_n^*(d)$ for $D(p_d^*) \leq d' < D(\tilde{p}_d)$.

The fact that $C_n^*(d)$ is a piecewise constant, right continuous function, as well as 2.a, follow from (1), (2) and (3).

2.b) Let $\mathcal{M}$ be the set of nodes $k \in N_+(n)$ for which it holds

$$C_k^*(d_0) = C_k^*(d_0 - d_{kn}) + c_{kn}.$$

It follows from Lemma 1 that $\mathcal{M}$ is non empty and

$$C_n^*(d_0) < C_k^*(d_0 - d_{kn}) + c_{kn}, \quad \text{for } k \in N_+(n) - \mathcal{M}$$

Since $C_n^*(d)$ is discontinuous at $d_0$, the continuity of the function $\min \{\bullet\}$ implies that for at least one node $m$ in $\mathcal{M}$, $C_n^*(d)$ is discontinuous at $d_0 - d_{mn}$.

3) According to 2), if $C_n^*(d)$ is discontinuous a $d_l$, there exist a path $p_d^*$ such that $D(p_d^*) = d_l$. Hence to different discontinuities (i.e., different $d_l$) correspond different paths and the statement follows from the fact that the number of paths in the network is finite.

The following observations that follow from the lemmas are important in the development of the algorithms below.

- Since $C_n^*(d)$ is piecewise constant, knowing its discontinuity points in fact determines the whole function.
- Let $C_n^*(d)$ be discontinuous at $d_0$. According to Lemma 2, 2.b, there is an $m \in N_+(n)$ such that $C_m^*(d)$ is discontinuous at $d_0 - d_{mn}$ and $C_m^*(d_0) = C_n^*(d_0 - d_{mn}) + c_{mn}$. Moreover, according to Lemma 2, 2.a, there is a path $q \in P_m(d_0 - d_{mn})$ such that $C_m^*(d_0 - d_{mn}) = C(q)$ and $D(q) = d_0 - d_{mn}$. Therefore, the path $p_d^*$ obtained by adjoining link $(m, n)$ to $q$, is a path with delay $d_0$ and cost $C_n^*(d_0)$. The triple $(d_0 - d_{mn}, C_m^*(d_0 - d_{mn}), m)$ is called predecessor of $(d_0, C_n^*(d_0), n)$. The successor of $(d_0 - d_{mn}, C_m^*(d_0 - d_{mn}), m)$.
- Suppose that we know that for $m \in N$, $C_n^*(d)$ is discontinuous at $d_0$ and in addition we know a path $q$ for which $C_m^*(d_0) = C(q)$, $d_0 = D(q)$. Then the possible successors of $(d_0, C_n^*(d_0), m)$ are the triplets $(d_0 + d_{mn}, C_m^*(d_0 + d_{mn}), n)$ for $n \in N_+(m)$. If we find a way of deciding which of these possible successors are actual ones, then we will immediately know the corresponding path by adjoining node $n$ to $p$.

In the following we also make use of the lexicographic order between pairs of real numbers. We say that the pair $(d_1, c_1)$ of real numbers is lexicographically smaller (or simply smaller if there is no possibility for confusion) than $(d_2, c_2)$ and write

$$(d_1, c_1) < (d_2, c_2),$$

if either $d_1 < d_2$, or $d_1 = d_2$ and $c_1 < c_2$.

### A. ALGORITHM 1

Let $D = \bigcup_{n \in N} D_n$ be the set of all discontinuities of the functions $C_n^*(d)$, $n \in N$. The proposed algorithm determines the discontinuities in $D$ in nondecreasing lexicographic order. Clearly, a smallest discontinuity occurs at $(0, 0)$ for the function $C_n(d)$. We assume the implementation of queues and heap structures with the following operations [1].

- **Queue structure** $Q$
  - `head(Q)`: returns (i.e., shows or points to, without removing) the first element $e$ of the $Q$.
  - `tail(Q)`: returns the last element $e$ of the $Q$.
  - `enqueue(Q, e)`: inserts element $e$ at the end of $Q$.
  - `dequeue(Q, e)`: removes and returns the head element $e$ of $Q$.
  - `size(Q)`: returns the size of $Q$.

  All previous operations on $Q$ take $O(1)$ time.

- **A heap $H$, using key $K$**
  - `create_heap(H)`: creates an empty heap $H$.
  - `insert(e, H)`: inserts element $e$ to $H$.
  - `find_min(H)`: returns an element $e$ in $H$ with the smallest key.
The outgoing neighbors of possible discontinuity of the form
(steps 5-7). The latter (possible discontinuities) consist of one
(steps 1-4). At this stage only the queue corresponding to the
mum key discontinuity
starts by initializing the queues
no possible discontinuities left to be examined.
We assume a Fibonacci heap implementation [4] of
Algorithm I finds all the discontinuities of the functions
Parameter discontinuity_node is of course redundant, but
The proposed algorithm is shown in Figure 1. The algorithm
starts by initializing the queues \( A_b[n] \), \( n \in N \) and the heap \( H_a \)
steps 1-4). At this stage only the queue corresponding to the
source node \( s \) is nonempty, containing the single discontinuity
at \((0,0)\) with null predecessor. The rest of the queues
are initialized to \((−∞, ∞, null, n)\). The latter initialization is
done in order to facilitate the description of the code. The heap
\( H_a \) contains the possible successor discontinuities of \((0,0,s)\)
(steps 5-7). The latter (possible discontinuities) consist of one
possible discontinuity of the form \((d_{sn}, c_{sn}, s, n)\) for each of
the outgoing neighbors of \( s \).
In the while loop, line 8, the algorithm removes the mini-
mum key discontinuity \( e_a \) among the possible discontinuities
in \( H_a \). Next, it compares the cost parameter of the key of
\( e_a \) with the cost parameter of the key of the tail element
\( e_b \) in the queue that corresponds to the discontinuity_node
of \( e_a \). If \( e_a.cost \) is larger than or equal to \( e_b.cost \), then
\( e_a \) is discarded. Else the discontinuity represented by \( e_a \)
is enqueued to the discontinuities of the queue corresponding to
\( m = e_a\text{discontinuity_node} \). Next, in the for all loop, line
14, a possible discontinuity is added to \( H_a \) for each outgoing
neighbors of \( m \). A further optimization step is taken here
by avoiding to create possible discontinuities for the node
\( n \in N−(m) \) for which \( n = e_a\text{predecessor} \), since such a
discontinuity is impossible.
The algorithm stops when \( H_a = \emptyset \), that is, when there are
no possible discontinuities left to be examined.
Algorithm I finds all the discontinuities of the functions
\( C_m^n(d), n \in N \), that is, all optimal paths from \( s \) to any node
\( n \in N \) and for any possible delay. If we are interested only
in finding a path from node \( s \) to a given node \( n \), with delay
at most \( d \), then the algorithm can be made to stop as soon
as for function \( C_m^n(d) \) a discontinuity is found whose delay is
larger than or equal to \( d \). Below we prove the correctness
of Algorithm I and analyze its computational complexity.
Correctness of Algorithm I
At the \( L^{th} \) iteration of the loop that begins on line 8, let
\( D_L = \cup_{n \in N} A_b[n] \). We will establish by induction that:
1) The \( L^{th} \) smallest discontinuity \( e_a \) in \( D \) is added to the
appropriate queue \( A_b[n] \).
2) The keys of the elements in heap \( H_a \) are larger than or
equal to the keys of the elements in \( D_L \).
3) The heap \( H_a \) contains all the discontinuities that may
be successors of the discontinuities in \( D_L \).
The statement is correct at the initialization step \( L = 0 \).
Assume next that the statement is correct at step \( L \). Let \( e_a \) be
the element with smallest key in \( H_a \) and \( e_b \) the last element in
\( A_b[m] \), where \( m = e_a\text{discontinuity_node} \). If \( e_a.cost \leq e_b.cost \),
then \( e_a \) cannot be a discontinuity of \( C_m^n(d) \). To see this, note
that by assumptions 1 and 2, we know that \( e_a.delay \geq e_b.delay \).
Therefore, for \( e_a \) to represent a discontinuity of \( C_m^n(d) \), by
Lemma 2 we must have \( e_a.cost > e_b.cost \). Assume now that
\( e_a.cost > e_b.cost \). Then, \( e_a.delay < e_b.delay \) since otherwise
the key of \( e_a \) will be (strictly) smaller than the key of \( e_b \),
which contradicts assumption 2. By assumption 3, there is
a predecessor of \( e_a \) in \( D_L \). There can be no discontinuity
\( e'_a \) for \( C_m^n(d) \) with delay smaller than \( e_a.delay \). To see this,
assume that such an \( e'_a \) exists. According to Lemma 2.b,
there must be a path \( p'_a = (m_1, m_2, ..., m_k) \), \( m_1 = s \)
\( m_k = e'_a\text{discontinuity_node} \), such that 1) \( C(p) = e'_a.cost \),
\( D(p) = e'_a.delay \) and 2) path \( p'_a = (m_1, ..., m_l) \), \( 1 \leq l \leq k-1 \)
corresponds to a discontinuity smaller than or equal to \( e'_a \).
Let \( m_l \) be the smallest index node such that the discontinuity

**Algorithm I**

*Inputs:* Graph \( G \) with link costs \( c_{ij} \) and
delays \( d_{ij} \). *Outputs:* The array \( A_b[n] \) of queues, which con-
tains the discontinuities of each node. */ begin initialization */

1. create_heap(\( H_a \))
2. for all \( n \in N - \{s\} \)
   
   3. \( A_b[n] = (−∞, ∞, null, n) \)
   
   4. \( A_b[s] = (0, 0, null, s) \)

   5. for all \( n \in N−(s) \)
   
   6. \( e_a = (d_{sn}, c_{sn}, s, n) \)

   7. insert(\( e_a, H_a \)); /* end initialization */

8. while \( H_a = \emptyset \) do
   
   9. get_min(\( e_a, H_a \));

   10. \( m = e_a\text{discontinuity_node} \);

   11. tail(\( e_a, A_b[m] \));

   12. if (\( e_b.cost > e_a.cost \)) then

   13.   enqueue(\( e_a, A_b[m] \));

   14.   for all \( n \in N−(m) \), \( n \neq e_a\text{predecessor} \)

   15.       \( e'_a = (e_a.delay + d_{mn}, e_a.cost + c_{mn}, m, n) \)

   16.       insert(\( e'_a, H_a \));

Fig. 1. Algorithm I
corresponding to $p^l_a = (m_1, \ldots, m_k)$ does not belong to $D_L$. The predecessor of $e_a$ must belong to $D_L$ and therefore, according to assumption 3, $e_a$ must belong to $H_a$. Since $e_a', \text{delay} \geq e_a', \text{delay}$, and $e_a', \text{delay} > e_a', \text{delay}$, we conclude that $e_a', \text{delay}$, which contradicts the fact that $e_a$ has the minimum key in $H_a$. Since by construction there is a path with delay and cost respectively $e_a', \text{delay}$ and $e_a', \text{cost}$, from the discussion above we conclude that $e_a$ represents a discontinuity of $C_n^a(d)$, $m = e_a', \text{discontinuity_node}$ and assumption 1 is satisfied for $L + 1$. Assumptions 2 and 3 are also satisfied since the insert operations in the for all loop, line 14, creates keys that are larger than or equal to $e_a$, as well as all the discontinuities that may be ancestors of $e_a$.

**Computational Complexity of Algorithm I.**

In the following we analyze the worst case running time of the algorithm provided that we intend to find all the discontinuities in $D$. A similar analysis holds for the worst case analysis when a bound $d$ is specified on the delays. We express these bounds in terms of parameters that are revealing of the performance of the algorithms in the average case. If desired, these bounds can also be expressed in terms of node, link numbers and the delay bound.

Let $R(n)$ be the number of discontinuities of $C_n^a(d)$. Denote,

$$R_{\text{max}} = \max_{n \in N} \{ R(n) \},$$

$$R_S = \sum_{n \in N} R(n),$$

$$E = \sum_{n \in N} |N_-(m)| R(m),$$

$$N_{\text{max}}^- = \max_{n \in N} \{ |N_-(m)| \}.$$  

Each discontinuity that is added to $A_0[m]$ inserts $|N_-(m)|$ elements to $H_a$. Therefore, $E$ elements are inserted in $H_a$ and its size is at most $E$. The get_min operation is executed once for each element of $H_a$. Since the get_min operation takes $O(\log E)$ time in the worst case, the worst case running time due to the get_min operation is $O(\log E)$. The insert operation takes constant time and is executed $E$ times. Hence the worst case running time of the algorithm is $O(E \log E)$.

We can express the worst case bound of the algorithm in terms of other relevant parameters as follows. Observe that

$$E \leq R_{\text{max}} \sum_{n \in N} |N_-(n)| = R_{\text{max}} |L|,$$

where $|L|$ is the cardinality of the set $L$. Hence the worst case performance of the proposed algorithm is,

$$O(R_{\text{max}} |L| (\log |L| + \log R_{\text{max}}))$$

Note that no integrality assumptions are made regarding the link delays. For comparison, assuming that the delays are positive integers and using the dynamic programming recursive equation in Lemma 1, the function $C_n^a(d)$ can be determined in the worst case running time, [6],

$$O(D_{\text{max}} |L|),$$

where $D_{\text{max}}$ is the maximum delay at which a discontinuity in $D$ may occur. If delays can take zero values, then the worst case running time of the dynamic programming recursive equations becomes

$$O(D_{\text{max}} (|L| + |N| \log |N|)).$$

As we will see in Section III, numerical results show that for a wide range of networks, $D_{\text{max}}$ is much larger than $R_{\text{max}}$. As a result, the running time of the proposed algorithm is in general significantly better than the algorithm obtained by a direct application of the dynamic programming equation.

**B. ALGORITHM II**

The performance of Algorithm I presented in the previous section can be improved by a more efficient organization of the heap $H_a$ containing all possible successors of the already known discontinuities. We present this approach in the current section.

Instead of the heap $H_a$ we consider the following structures.

- An array $B[l]$ of queues $l \in L$. An element $e$ of $B[l]$ is of the form,

$$e = (\text{delay}, \text{cost}, \text{predecessor}, \text{discontinuity_node}),$$

where $m=\text{predecessor}$ is the origin of link $l$, $n=\text{discontinuity_node}$ is the destination of link $l$ and $(\text{delay}, \text{cost})$ signifies a possible discontinuity of $C_n^a(d)$ with predecessor node $m$. As in the previous section, $n$ and $m$ are redundant here, but we keep them for simplicity in the description. Hence queue $B[l]$ contains all possible discontinuities of $C_n^a(d)$ that may be successors of the already known discontinuities of $C_n^a(d)$. The elements in $B[l]$ are stored in increasing order of keys, where key is the pair $(\text{delay}, \text{cost})$.

- An array of heaps $H_a[n]$. Heap $H_a[n]$ contains the head elements of the queues $B[(m,n)]$, $m \in N_+(n)$. An element $e_a$ of $H_a[n]$ is of the same form as the elements in array $B[l]$.

- A heap $H_g$ containing the minimum key elements of $H_a[n]$, $n \in N$. An element $e_g$ of $H_g$ is of the same form as the elements in array $B[l]$. Operations performed during the update of $H_g$ ensure that each element $e$ in $H_g$ has smaller delay than any of the discontinuities in $H_a[n]$, $n = e.\text{discontinuity_node}$. This, combined with the correctness proof of Algorithm I, ensures that the minimum key element $e_g$ in $H_g$ is a real discontinuity for $C_n^a(d)$, $n = e_g.\text{discontinuity_node}$.

We also need the following subroutines

- **obtain_minimum($e_g, H_g$)**: this subroutine returns the minimum-key element $e_g$ in $H_g$. At the same time, it updates $H_a[n]$, $n = e_g.\text{discontinuity_node}$ and $B[l]$, $l = (m,n)$, $m \in N_+(n)$, and either removes or updates element $e_g$.

- **update_new($e_a$)**: This subroutine inserts a possible discontinuity $e_a$ in queue $B[(m,n)]$, where $m = e_a.\text{predecessor}$ and $n = e_a.\text{discontinuity_node}$. At the same time, $H_a[n]$ and $H_g$ are updated.

The modified algorithm is presented in Figure 2. It is assumed without loss of generality that $N_+(s) = \emptyset$. 

0-7803-7753-2/03/$17.00 (C) 2003 IEEE IEEE INFOCOM 2003
Algorithm II Inputs: Graph G with link costs \( c_e \) and delays \( d_{ij} \). Outputs: The array \( A_b[n] \) of queues, which contains the discontinuities of each node. /* begin initialization */
1. create_heap(\( H_g \));
2. \( A_b[s] = (0, 0, null, s) \);
3. for all \( l \in L \)
   4. \( B[l] = \emptyset \);
5. for all \( a \in N - \{ s \} \)
   6. create_heap(\( H_a[n] \));
7. \( A_b[n] = (\emptyset, \emptyset, null, n) \);
8. for all \( a \in N - \{ s \} \)
   9. \( e_a = (d_{aa}, c_{mn}, s, n) \);
10. update_new(\( e_a \)); /* end initialization */
11. while \( H_g \neq \emptyset \) do
12. obtain_minimum(\( e_g, H_g \));
13. \( m = e_g.discontinuity_node \);
14. enqueue(\( e_g, A_b[m] \));
15. for all \( n \in N - \{ m \} \)
   16. \( e'_a = (d_{an}, c_{mn}, s, n) \);
17. update_new(\( e'_a \));

The initialization of \( A_b[n] \) with a single element \( (\emptyset, \emptyset, null, n) \) is made in order to simplify the description of the code.

It is instructive at this point to compare Algorithm II with Dijkstra’s algorithm [1, page 109]. In the latter algorithm, at each step there is a set \( S \subseteq N \) that consists of the nodes whose minimum distance from the source node is known, and there is a label associated with each node representing the shortest distance path from the node to the source, provided that only the nodes in \( S \) can be used as intermediate nodes in a path. At each step of the algorithm, a node \( n \) in \( S = N - S \) with smallest label is moved to \( S \) and the labels of all outgoing neighbors of \( n \) are updated.

There is a direct correspondence between Dijkstra’s algorithm and Algorithm II as follows

- \( S \) corresponds to the union of the elements in \( A_b[n] \), \( n \in N \).
- \( N \) corresponds to the set \( D \) of discontinuities of the functions \( C^*_n (d) \).
- The label of node \( n \) corresponds to the set of possible discontinuities of \( n \) that are located in \( B((m, n)), m \in N_+(n) \).

In this sense, we may say that the proposed algorithm is a generalization of Dijkstra’s algorithm.

We present the subroutine update_new in Figure 3, and we describe the various steps of the pseudocode.

If the cost of \( e_a \) is larger than or equal to the cost of the tail element \( e_t \) in \( A_b[n] \) (line 4), then \( e_a \) is not a possible discontinuity and therefore it is discarded. If the cost of \( e_a \) is smaller than the cost of \( e_t \), then, as with Algorithm I, it is known that \( e_a.delay > e_t.delay \). If the heap \( H_a[n] \) is empty, then the new element is inserted in \( B[l] \). \( H_a[n] \) and \( H_g \) (lines 5 to 7) and the subroutine ends. Else, (lines 8 and below) the key of \( e_a \) is compared with the minimum-key element \( e_{\min} \) in \( H_a[n] \) in order to decide whether \( e_a \) can be a possible discontinuity of \( C^*_n (d) \) and whether \( e_{\min} \) should be updated and certain possible discontinuities can be discarded. Specifically,

1. If \( e_{\min}.delay \leq e_a.delay \) and \( e_{\min}.cost \leq e_a.cost \) (line 10), then \( e_a \) cannot be a possible discontinuity of \( C^*_n (d) \) and therefore it is discarded.
2. If \( e_{\min}.delay < e_a.delay \) and \( e_{\min}.cost > e_a.cost \) (line 11), then \( e_a \) is a possible discontinuity and in case \( B[l] \) is empty, \( e_a \) must be inserted in \( H_a[n] \). Moreover, \( e_a \) is added to \( B[l] \).
3. If \( e_{\min}.delay > e_a.delay \) and \( e_{\min}.cost < e_a.cost \) (line 14), then \( e_a \) is a possible discontinuity. Since the key of \( e_a \) is smaller than the key of \( e_{\min} \), \( B[l] \) must be empty (the key of \( e_a \) is always larger than the key of the head element in \( B[l] \) which in turn is larger than or equal to the key of \( e_{\min} \)). Therefore \( e_a \) must be added to \( B[l] \), must be inserted in \( H_a[n] \) and must replace \( e_{\min} \) in \( H_g \).
4. If \( e_{\min}.delay = e_a.delay \) and \( e_{\min}.cost > e_a.cost \) or...
(e_{min} - delay > e_a - delay and e_{min} - cost ≥ e_a - cost) (line 17) then e_a is a possible discontinuity while e_{min} is not. Therefore, e_a replaces e_{min} both in H_a[n] and in H_g (lines 18, 19) and e_{min} is removed from B((k, n)), k = e_{min} - discontinuity_node (line 23). Next, (line 24) the queue B[(n, k)], is scanned and all impossible discontinuities in this queue are removed.

Finally, we present the obtain_minimum(e_g, H_g) subroutine in Figure 4.

Initially, the minimum-key element e_g in H_g is obtained. Next, e_g is dequeued from queue B[l] and H_a[n], where l = (m, n), n = e_g - discontinuity_node, m = e_g - predecessor, and if B[l] is not empty, the head element in B[l] is inserted in H_a[n]. The rest of the pseudocode (starting from line 10) determines the element in H_a[n] that is going to replace e_g and removes from queues B[(k, n)], k ∈ N_+ (n) impossible discontinuities. The minimum-key element e_{min} in H_a[n] is determined. In order to be inserted to H_g, e_{min} must have strictly larger delay and strictly smaller cost than e_g. If this is true (line 12), then e_{min} is inserted in H_g in place of e_g and the subroutine ends. Otherwise, since it is always true that the keys of the elements in H_a[n] are larger than or equal to the key of e_g, the only possibilities are that e_g - delay ≤ e_{min} - delay and e_g - cost ≤ e_{min} - cost. In such a case, e_{min} cannot be a possible discontinuity of C_n^+(d) and must be removed. The rest of the code in the while loop performs this removal and at the same time removes impossible discontinuities from B[(k, n)], where k = e_{min} - predecessor. On exit from the while loop (line 26), H_a(n) must be empty and therefore e_g is removed from H_g instead of being replaced with another element in H_a(n) as was done in lines 13, 23.

**Computational Complexity of Algorithm II**

We use the same notation as in Section II-A. Subroutine enqueue(e_g, A(m)) takes O(1) time and is executed once for each discontinuity in D. Subroutine obtain minimum() is invoked once for each discontinuity in D, that is R_S times. Each such invocation involves a get_min or increase_key operation on H_g, which takes O(log |N|) time in the worst case. Hence, the worst case running time for all these operations is O(R_S log |N|). There are also other computations involving H_a[n] and B[l], which are taken into account below.

Subroutine update_new() is invoked once per discontinuity in D. Each invocation causes |N_+ (m)| updates, to H_a[m] and B(m, n) corresponding to the outgoing neighbor m of the node n = e_a - discontinuity_node. These latter updates involve in the worst case an enqueue and dequeue operation to one of the queues in B[l], a get_min, insert, or decrease_key operation on H_a[m] and decrease_key operation on H_g. The get_min operations on H_a[m] take worst-case time O(log |N_+ (m)|) since the size of H_a[m] is at most |N_+ (m)|, while the rest of the operations take worst-case time O(1).

Hence, the total worst case running time of the algorithm is

$$O \left( R_S \log |N| + \sum_{n \in N} R(n) \sum_{m \in N_+ (n)} \log |N_+ (m)| \right),$$

Since

$$R_S \leq |N| R_{max},$$

and, denoting $N_{max}^+$ = max_m ∈ N \{|N_+ (m)|\},

$$\sum_{n \in N} R(n) \sum_{m \in N_+ (n)} \log |N_+ (n)| \leq R_{max} \log \left( N_{max}^+ \right) \sum_{n \in N} \sum_{m \in N_+ (n)} 1 = R_{max} \log \left( N_{max}^+ \right) |L|,$$

can we express the previous bound as

$$O \left( R_{max} \left(|N| \log |N| + |L| \log N_{max}^+ \right) \right).$$

Note that if all delays are zero, then we have in effect the unconstrained shortest path routing problem. In this case, each $C_n^+(d)$ has a single discontinuity at 0, and $H_a[n]$ never has more than a single element. That means that the factor $N_{max}^+$ can be removed from (5) and therefore the worst-case running time in this case becomes

$$O \left(|N| \log |N| + |L| \right),$$
that is, identical to the worst-case running time of Dijkstra’s algorithm. This is to be expected since in this case Algorithm II reduces in effect to Dijkstra’s algorithm.

In table I we summarize the worst-case running times of Algorithms I, II (ALG I and ALG II respectively) and the algorithm that results from the direct application of the dynamic programming equation (DP) for the given network, $D_{\text{max}}$ corresponds to the delay $d$ that causes the longest running time for all three algorithms tested.

We generated random uniform and power law networks with 400, 800, and 1200 nodes, and with ratios $\alpha = |L| / |N|$ equal to 4, 8, 16, which are close to or larger than the ratios between 3 and 4 commonly found in today’s networks. For each $|N|$ and $\alpha$, we generated 10 random networks. For each of the generated networks we created link costs according to COST 1 and COST 2. In tables II - V we present the parameters of the generated networks. The parameter values are the averages of the values obtained for each of the 10 random networks. We observe that in general $R_{\text{max}}$ is much smaller than $D_{\text{max}}$. For the same method of link cost generation, the variations of parameter values for random and power law networks, (with the exception of $N_{\text{max}}^+\alpha$) are not significant. However, networks where the COST 2 method is used for link cost generation generally have larger $R_{\text{max}}$ and $D_{\text{max}}$ than corresponding networks where the COST 1 method is used. As we will see, these observations have an effect on the performance of all three algorithms considered.

The tested Real Network consisted of 6474 nodes with 12572 bidirectional links, therefore $a = 3.88 (2x12572/6474)$. Link delays are again picked randomly with uniform distribution among the integers [1, 100]. Both COST 1 and COST 2 methods for generating link costs were tested. In this network, we performed 10 experiments, where in each experiment a node is picked randomly to represent the source. The quantity $D_{\text{max}}$ is defined in the same way as with the randomly generated networks. Table VI shows the relevant parameters in these experiments.

The experiments were run on a Pentium PC IV, 1.7GHz. The average running times (in seconds) for the three algorithms and for the various experiments are shown in tables VII - XI. Also, in these tables we present the ratio of the average running time of the DP algorithm to the average running time of ALG I and ALG II. The following observation are worthwhile.

- For the same link cost generation method, the variations of running times of a given algorithm for Random and Power Law networks with fixed $|N|$ and $\alpha$ are not significant.
- For a given algorithm, link costs generated by method COST 2 induce longer running times than link costs generated by method COST 1, for fixed $|N|$ and $\alpha$.
- The performance of ALG I is comparable to ALG II for $\alpha = 4$, however for larger values the running time of ALG II can be about two times shorter than that of ALG I.
- For all experiments the running times of ALG I and ALG II are significantly better than DP (18 to 100 times shorter).

It is also worth noting that the performance of DP algorithm depends on the desired accuracy of link delays, while this is not true for both ALG I and ALG II. For example, we picked link delays between 1 and 100, which implies that we require two-digit accuracy. If the required accuracy is increased, then both $D_{\text{max}}$ and $R_{\text{max}}$ may increase, but the increase in $R_{\text{max}}$ is much slower, and therefore the difference in the running times of the algorithms becomes even more significant.

III. Numerical Results

We run three sets of experiments. Each set corresponds to different methods of network generation, as follows.

- **Uniform Networks**: A number $|N|$ of nodes and a number $|L| = \alpha |N|$ of edges, $\alpha > 1$ is chosen. We use the graph generator random_graph( ) from the LEDA package [12]. A random edge is generated by selecting a random element from a candidate set $C$ defined as follows:
  - $C$ is initialized to the set of all $|N| (|N| - 1)$ pairs $(u, w)$ of distinct nodes.
  - Upon a selection of a pair $(u, w)$ from $C$, the pair is removed from $C$.

  For each edge, a delay is picked randomly with uniform distribution among the integers [1, 100].

- **Power Law Networks**: A number $|N|$ of nodes and a number $|L| = \alpha |N|$, $\alpha > 1$ of links are chosen. The $|L|$ links are used to connect nodes randomly in such a manner that the node degrees follow a power law [16]. This is one of the methods that attempt to generate network topologies that are “Internet like”. The nodes are placed randomly on a grid and the link delays are set to be proportional to the distance between the nodes joined by the link under consideration.

- **Real Internet Network Topology**: This network topology was taken from [17] and is based on the network topology observed in 01/02/2000. For each edge, a delay is picked randomly with uniform distribution among the integers [1, 100].

  For all the experiments, link costs are generated by one of the following methods.

- **COST 1**: A cost $c_l$ for link $l$ is picked randomly with uniform distribution among the integers [1, 100].

- **COST 2**: A parameter $\sigma_l$ for link $l$ is picked randomly with uniform distribution among the integers [1, 5]. The cost for the link under consideration is then $c_l = \sigma_l (101 - d_l)$. This method of cost generation reflects the situation where link costs are decreasing as link delays are increasing.

  For each of the experiments, we determine all discontinuities of the functions $C_n^*(d)$, $n \in N$, that is, in effect we find all optimal paths from a node $s$ to all nodes in the network, under any possible delay constraint. We denote by $D_{\text{max}}$ the maximum delay at which discontinuities in $D$ occur. If one is interested only in finding a path with delay at most $d$, then
<table>
<thead>
<tr>
<th>N</th>
<th>400</th>
<th>800</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>$D_{\text{max}}$</td>
<td>679.7</td>
<td>679.1</td>
<td>558.9</td>
</tr>
<tr>
<td>$R_{\text{max}}$</td>
<td>11.7</td>
<td>16.1</td>
<td>18.8</td>
</tr>
<tr>
<td>$N_{\text{max}}^+$</td>
<td>10.9</td>
<td>16.8</td>
<td>28.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>400</th>
<th>800</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>$D_{\text{max}}$</td>
<td>516.3</td>
<td>485</td>
<td>453</td>
</tr>
<tr>
<td>$R_{\text{max}}$</td>
<td>12.9</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>$N_{\text{max}}^+$</td>
<td>48</td>
<td>68</td>
<td>94</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>400</th>
<th>800</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>$D_{\text{max}}$</td>
<td>899.8</td>
<td>1027.2</td>
<td>1073</td>
</tr>
<tr>
<td>$R_{\text{max}}$</td>
<td>19.4</td>
<td>35</td>
<td>61.9</td>
</tr>
<tr>
<td>$N_{\text{max}}^+$</td>
<td>10.9</td>
<td>16.8</td>
<td>28.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>400</th>
<th>800</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>$D_{\text{max}}$</td>
<td>710</td>
<td>833</td>
<td>839</td>
</tr>
<tr>
<td>$R_{\text{max}}$</td>
<td>21.7</td>
<td>37.8</td>
<td>86.4</td>
</tr>
<tr>
<td>$N_{\text{max}}^+$</td>
<td>48</td>
<td>68</td>
<td>94</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>400</th>
<th>800</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>$D_{\text{max}}$</td>
<td>776</td>
<td>1001</td>
<td>1073</td>
</tr>
<tr>
<td>$R_{\text{max}}$</td>
<td>26</td>
<td>93</td>
<td>26</td>
</tr>
<tr>
<td>$N_{\text{max}}^+$</td>
<td>1458</td>
<td>1458</td>
<td>1458</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>400</th>
<th>800</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>ALG I</td>
<td>0.0593</td>
<td>0.214</td>
<td>0.625</td>
</tr>
<tr>
<td>ALG II</td>
<td>0.0579</td>
<td>0.1485</td>
<td>0.3296</td>
</tr>
<tr>
<td>DP</td>
<td>5.875</td>
<td>11.029</td>
<td>17.56</td>
</tr>
<tr>
<td>DP/ALG I</td>
<td>99.077</td>
<td>51.5</td>
<td>28.1</td>
</tr>
<tr>
<td>DP/ALG II</td>
<td>101.4</td>
<td>74.27</td>
<td>53.29</td>
</tr>
</tbody>
</table>
TABLE VIII
AVERAGE RUNNING TIMES FOR RANDOM POWER LAW NETWORKS, COST 1

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>400</th>
<th>800</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALG I</td>
<td>0.049</td>
<td>0.183</td>
<td>0.529</td>
</tr>
<tr>
<td>ALG II</td>
<td>0.065</td>
<td>0.143</td>
<td>0.298</td>
</tr>
<tr>
<td>DP</td>
<td>4.35</td>
<td>7.56</td>
<td>13.58</td>
</tr>
<tr>
<td>DP/ALG I</td>
<td>87.3</td>
<td>41.2</td>
<td>25.6</td>
</tr>
<tr>
<td>DP/ALG II</td>
<td>66.4</td>
<td>52.5</td>
<td>45.5</td>
</tr>
</tbody>
</table>

TABLE IX
AVERAGE RUNNING TIMES FOR RANDOM UNIFORM NETWORKS, COST 2

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>400</th>
<th>800</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALG I</td>
<td>0.0845</td>
<td>0.3858</td>
<td>1.276</td>
</tr>
<tr>
<td>ALG II</td>
<td>0.0826</td>
<td>0.2577</td>
<td>0.6188</td>
</tr>
<tr>
<td>DP</td>
<td>7.91</td>
<td>17.098</td>
<td>34.88</td>
</tr>
<tr>
<td>DP/ALG I</td>
<td>93.6</td>
<td>44.31</td>
<td>27.32</td>
</tr>
<tr>
<td>DP/ALG II</td>
<td>95.77</td>
<td>66.35</td>
<td>56.37</td>
</tr>
</tbody>
</table>

TABLE X
AVERAGE RUNNING TIMES FOR RANDOM POWER LAW NETWORKS, COST 2

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>400</th>
<th>800</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALG I</td>
<td>0.077</td>
<td>0.346</td>
<td>1.11</td>
</tr>
<tr>
<td>ALG II</td>
<td>0.101</td>
<td>0.262</td>
<td>0.583</td>
</tr>
<tr>
<td>DP</td>
<td>6.12</td>
<td>13.4</td>
<td>26.1</td>
</tr>
<tr>
<td>DP/ALG I</td>
<td>78.6</td>
<td>38.7</td>
<td>23.5</td>
</tr>
<tr>
<td>DP/ALG II</td>
<td>60.3</td>
<td>51.1</td>
<td>44.7</td>
</tr>
</tbody>
</table>

TABLE XI
AVERAGE RUNNING TIMES FOR REAL INTERNET NETWORK

<table>
<thead>
<tr>
<th>COST 1</th>
<th>COST 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALG I</td>
<td>2.641</td>
</tr>
<tr>
<td>ALG II</td>
<td>2.765</td>
</tr>
<tr>
<td>DP</td>
<td>132.15</td>
</tr>
<tr>
<td>DP/ALG I</td>
<td>50</td>
</tr>
<tr>
<td>DP/ALG II</td>
<td>47.8</td>
</tr>
</tbody>
</table>

demonstrate this we run the following experiment. We generate a Uniform network with $|N| = 800$ nodes and $|L| = \alpha |N| = 8 \times 800 = 6400$ edges. In the first experiment we pick delays between $[1,100]$ and in the second between $[1,1000]$. In table XII we present the relevant parameters and the running times for the tested network. We observe that the running time of DP algorithm increases tenfold while the increases of the running times of ALG I and ALG II are insignificant.

IV. CONCLUSIONS

In this paper we addressed the QoS routing problem. We provided two algorithms for finding the optimal solution. The basic idea of the proposed algorithms consists in finding in an iterative fashion the discontinuities of the functions $C^*_n(d)$.

The algorithms operate under nonnegative link costs and delays and do not require any integrality assumptions on the delays. Numerical results show that for a wide range of tested networks the proposed algorithm outperforms significantly the algorithm based on the direct implementation of the Dynamic Programming equations. The running time of the proposed algorithms is satisfactory even for relatively large network sizes. If worst case performance is also of concern, the algorithms can replace the DP recursions of the approximate polynomial-time algorithms proposed in [9] and [6], in order to improve their average running time. The algorithms can be extended to the case where multiple constraints on the paths exist (e.g., maximum delay and loss probabilities).

Acknowledgement 1: The code for generating Power Law
TABLE XII
PARAMETERS AND RUNNING TIMES FOR A RANDOM UNIFORM NETWORK, COST 2

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value 1</th>
<th>Value 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>800</td>
<td></td>
</tr>
<tr>
<td>delay</td>
<td>(1,100)</td>
<td>(1,1000)</td>
</tr>
<tr>
<td>D&lt;sub&gt;max&lt;/sub&gt;</td>
<td>1425</td>
<td>16611</td>
</tr>
<tr>
<td>R&lt;sub&gt;max&lt;/sub&gt;</td>
<td>42</td>
<td>45</td>
</tr>
<tr>
<td>N&lt;sub&gt;max&lt;/sub&gt;</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>ALG I</td>
<td>0.797</td>
<td>0.922</td>
</tr>
<tr>
<td>ALG II</td>
<td>0.531</td>
<td>0.594</td>
</tr>
<tr>
<td>DP</td>
<td>49.26</td>
<td>599.64</td>
</tr>
<tr>
<td>DP/ALG I</td>
<td>61.8</td>
<td>650.3</td>
</tr>
<tr>
<td>DP/ALG II</td>
<td>92.7</td>
<td>1009.5</td>
</tr>
</tbody>
</table>

Networks was downloaded from site [16], and the data for the Real Network from [17]. We would like to thank the authors of these sites.

REFERENCES